1 INTRODUCTION

> *Jealous stepmother and sisters; magical aid by a beast; a marriage won by gifts magically provided; a bird revealing a secret; a recognition by aid of a ring; or show; or what not; a dénouement of punishment; a happy marriage - all those things, which in sequence, make up Cinderella, may and do occur in an incalculable number of other combinations.*
>
> — MR. Cox ***1893***, *Cinderella: Three hundred and forty-five variants [**?**]*

This dissertation presents toolsets, approaches and methodologies designed to enhance WebAssembly security proactively through Software Diversification. First, Software Diversification could expand the capabilities of the mentioned tools by incorporating diversified program variants, making it more challenging for attackers to exploit any missed vulnerabilities. Generated as proactive security, these diversified variants can simulate a broader set of real-world conditions, thereby making WebAssembly analysis tools more accurate. Second, we noted that current solutions to mitigate side-channel attacks on WebAssembly binaries are either specific to certain attacks or need the modification of runtimes, e.g., Swivel as a cloud-deployed compiler. Software Diversification could mitigate yet-unknown vulnerabilities on WebAssembly binaries by generating diversified variants in a platform-agnostic manner.

Our work focuses on artificial diversification techniques, particularly for WebAssembly, drawing insights from significant works by Baudry et al. [**?**] and Jackson et al. [**?**].

TODO From the LIC

The first web browser, Nexus [**?**], appeared in 1990. At that moment, web browsing was only about retrieving and showing small and static HTML web pages. In other words, users read the content of pages without interactions. The growing computing power of devices, the spread of the internet, and the need for more interaction and experiences for users encouraged the idea of executing code along with web pages. The Netscape browser made possible the execution of code on the client-side with the introduction of the JavaScript [**?**] language in 1995. Remarkably, all browsers have supported JavaScript since Netscape. Nowadays,

[0] Compilation probe time 2023/10/25 10:48:51

most web pages include not only HTML, but also include JavaScript code that is executed in client computers. During the past decades, web browsers have become JavaScript language virtual machines. They evolved to complex systems that can run full-fledged applications, like video and audio players, animation creators, and PDF document renderers such as the one showing this document.

JavaScript is currently the most used scripting language in all modern web browsers [?]. However, JavaScript faces several limitations related to the characteristics of the language. For example, any JavaScript engine requires the parsing and recompiling the JavaScript code, which implies a significant overhead. Moreover, JavaScript faces security issues [?]. For example, JavaScript lacks of memory isolation, making possible to extract pieces of information from others processes [?]. Because of these problems, the Web Consortium (W3C) standardized in 2017 a bytecode for the web environment, the Wasm (Wasm) language.

Wasm is designed to be fast, portable, self-contained, and secure [?]. All WebAssembly programs are compiled ahead-of-time from source languages such as C/C++ and Rust. Wasm is created by third-party compilers that might include optimizations like in the case of LLVM. The Wasm language defines its Instruction Set Architecture [?] as an abstraction close to machine code instructions but agnostic to CPU architectures. Thus, web browsers can use it to rapidly compile to the target architectures in a one-to-one translation process.

WebAssembly evolved outside web browsers. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup and memory usage than containerization and virtualization [? ?]. Consequently, in 2019, the Bytecode Alliance [?] proposed WebAssembly System Interface (WASI) [?]. WASI pionered the execution of Wasm with a POSIX system interface protocol, making possible to execute Wasm directly in the operating system. Therefore, it standardizes the adoption of Wasm in heterogeneous platforms [?], making it suitable for edge-cloud computing platforms [? ?]

1.1 Software Monoculture

Web browsers and JavaScript have nearly three decades of development. Since then, web browsers have grown, reaching several implementations [? ?]. Nevertheless, only Firefox, Chrome, Safari, and Edge dominate on user computers. This means that, for 5 arbitrary devices (computers, tablets, smartphones) in a world of millions, at least two of them use the same web browser. This highlights a software monoculture problem [?], as an ecosystem of machines running the exact same software. The monoculture concept is an analogy from biology [?]. It describes an ecosystem that faces extinction due the lack of diversity as all individuals share the exact same vulnerabilities. In other words, many applications can crash due to a single shared vulnerability.

Nowadays, the serving of web pages, including WebAssembly code, is centralized and provided through main servers [**?**]. Thus, a similar argument for software monoculture can be used for the Wasm code that is served to web browsers. Despite being designed for sandboxing and secure execution, Wasm is not exempt from vulnerabilities [**?**]. For example, Wasm engines are vulnerable to speculative execution [**?**], and C/C++ source code vulnerabilities might be ported to Wasm binaries [**?**]. Therefore, the sharing of the Wasm code through web browsers, also includes Wasm vulnerabilities.

The software monoculture problem escalates if we consider the edge-cloud computing platforms and how they are adopting Wasm to provide services, as we previously mentioned. Concretely, along with browser clients, thousands of edge devices running Wasm as backend services might be affected due to vulnerabilities sharing. This means that if one node in an edge network is vulnerable, all the others are vulnerable in the exact same manner as the same binary is replicated on each node. In other words, the same attacker payload would break all edge nodes at once. This illustrates how Wasm execution is fragile with respect to systemic vulnerabilities for the whole internet. Let us take the example of what happened on June 8, 2021, with Fastly [**?**]. That day, the whole internet suffered a 45 minutes disruption because of a failure when one Wasm binary was deployed at Fastly. The complete Fastly platform crashed. The bug, combined with most web pages being CDN-dependant, created a catastrophe. Therefore, a single distributed Wasm binary could unleash the same incident [**?**].

One might think that the solution is to adopt more web browser and interpreters implementations. However, this is virtually impossible as 4 web browsers dominate the market and edge-cloud computing platforms are transparently executed in the backend. Thus, a solution in this direction is doomed to fail. Another solution is to provide different WebAssembly codes. For example, a different source code, yet equivalent, can be provided when a web page requests it [**?**]. Consequently, millions of computers would execute different codes even though they use the same web browser. This strategy is called Software Diversification.

1.2 Software Diversification

Software Diversification is the process of finding, creating, and deploying program variants of a given original program [**?**] for the sake of security. Cohen et al. [**?**] and Forrest et al. [**?**] pioneered this field by proposing software diversification through code transformations. They proposed to produce variants of programs while preserving their functionalities, aiming to mitigate vulnerabilities. Since then, transformations aiming at reducing the predictability of observable behavior of programs have been proposed. For example, works on this direction proposed to diversify programs control flow [**?**], instruction set [**?**], or the system calls

they use [**?**]. Several of these transformations can be combined to produce less predictable variants.

While previous works on software diversification demonstrated the removal of vulnerabilities, in all cases, it can be used as a preemptive solution. For example, if a vulnerability is present in one program variant, discovering and disseminating it will not affect other variants. Software diversification has been widely researched, yet, the field does not study its application to Wasm. Only Romano et al. [**?**] proposed the intermixing JavaScript and Wasm function calls to provide obfuscation against code analysis. For Wasm, no software diversification solution has been proposed, primarily due to its novelty.

TODO Recent papers first. Mention Workshops instead in conference. "Procedings of XXXX". Add the pages in the papers list.

1.3 Background

TODO Motivate with the open challenges.

1.4 Problem statement

TODO Problem statement **TODO** Set the requirements as R1, R2, then map each contribution to them.

1.5 Automatic Software diversification requirements

1. **1:** **TODO** Requirement 1

1.6 List of contributions

In the space of Software Diversification we make the following contributions.

C1 Experimental contribution: For each proposed technique we provide an artifact implementation and conduct experiments to assess its capabilities. The artifacts are publicly available. The protocols and results of assessing the artifacts provide guidance for future research.

C2 Theoretical contribution: We propose a theoretical foundation in order to improve Software Diversification for WebAssembly.

C3 Diversity generation: We generate WebAssembly program variants.

Contribution	Research papers			
	P1	P2	P3	P4
C1 Experimental contribution	✓	✓	✓	✓
C2 Theoretical contribution	✓		✓	
C3 Diversity generation	✓	✓	✓	✓
C4 Defensive diversification	✓	✓	✓	
C5 Offensive diversification				✓

Table 1.1

C4 Defensive Diversification: We assess how generated WebAssembly program variants could be used for defensive purposes.

C5 Offensive Diversification: We assess how generated WebAssembly program variants could be used for offensive purposes, yet improving security systems.

1.7 Summary of research papers

This compilation thesis comprises the following research papers.

P1: **CROW: Code randomization for WebAssembly bytecode.**
Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
Measurements, Attacks, and Defenses for the Web (MADWeb 2021), 12 pages
`https://doi.org/10.14722/madweb.2021.23004`

Summary: In this paper, we introduce the first entirely automated workflow for diversifying WebAssembly binaries. We present CROW, an open-source tool that implements software diversification through enumerative synthesis. We assess the capabilities of CROW and examine its application on real-world, security-sensitive programs. In general, CROW can create statically diverse variants. Furthermore, we illustrate that the generated variants exhibit different behaviors at runtime.

P2: **Multivariant execution at the Edge.**
Javier Cabrera-Arteaga,Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Moving Target Defense (MTD 2022), 12 pages
`https://dl.acm.org/doi/abs/10.1145/3560828.3564007` **Summary:**

In this paper, we synthesize functionally equivalent variants of a deployed

edge service. These variants are encapsulated into a single multivariant WebAssembly binary, When executing the service endpoint, a random variant is selected each time a function is invoked. Execution of these multivariant binaries occurs on the global edge platform provided by Fastly, as part of a research collaboration. We demonstrate that these multivariant binaries present a diverse range of execution traces throughout the entire edge platform, distributed worldwide, effectively creating a moving target defense.

P3: **Wasm-mutate: Fast and efficient software diversification for WebAssembly.**
Javier Cabrera-Arteaga, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry
Under review, 17 pages
`https://arxiv.org/pdf/2309.07638.pdf`

Summary: This paper introduces WASM-MUTATE, a compiler-agnostic WebAssembly diversification engine. The engine is designed to swiftly generate semantically equivalent yet behaviorally diverse WebAssembly variants by leveraging an e-graph. We show that WASM-MUTATE can generate tens of thousands of unique WebAssembly variants in mere minutes. Importantly, WASM-MUTATE can safeguard WebAssembly binaries from timing side-channel attacks, such as Spectre.

P4: **WebAssembly Diversification for Malware evasion.**
Javier Cabrera-Arteaga,Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023, 17 pages
Summary: WebAssembly, while enhancing rich applications in browsers, also proves efficient in developing cryptojacking malware. The rise of cryptojacking malware has sparked numerous protective measures. However, these defenses have not factored in the potential use of evasion techniques by attackers. This paper delves into the potential of automatic binary diversification in aiding WebAssembly cryptojacking detectors' evasion. We provide proof that our diversification tools can generate variants of WebAssembly cryptojacking that successfully evade VirusTotal and MINOS. We further demonstrate that these generated variants introduce minimal performance overhead, thus verifying binary diversification as an effective evasion technique.

■ Thesis layout

This dissertation comprises two parts as a compilation thesis. Part one summarises the research papers included within, which is partially rooted in the author's licentiate thesis [**?**]. Chapter 2 offers a background on WebAssembly

and the latest advancements in Software Diversification. Chapter 3 delves into our technical contributions. Chapter 4 exhibits two use cases applying our technical contributions. Chapter 5 concludes the thesis and outlines future research directions. The second part of this thesis incorporates all the papers discussed in part one.